PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : G06F 9/00

A2

(11) Internati nal Publication Number: **WO 00/00885**

(43) International Publication Date: 6 January 2000 (06.01.00)

(21) International Application Number: PCT/US99/13896

(22) International Filing Date: 22 June 1999 (22.06.99)

(30) Priority Data:
09/107,382 30 June 1998 (30.06.98) US

(71) Applicant: SUN MICROSYSTEMS, INC. [US/US]; 901 San Antonio Road, MS UPAL01 – 521, Palo Alto, CA 94303 (US).

(72) Inventor: WOLCZKO, Mario; 580 Arastradero Road #503, Palo Alto, CA 94306 (US).

(74) Agents: GARRETT, Arthur, S.; Finnegan, Henderson, Farabow Garrett & Dunner, L. L.P., 1300 I Street, N.W., Washington, DC 20005–3315 (US) et al.

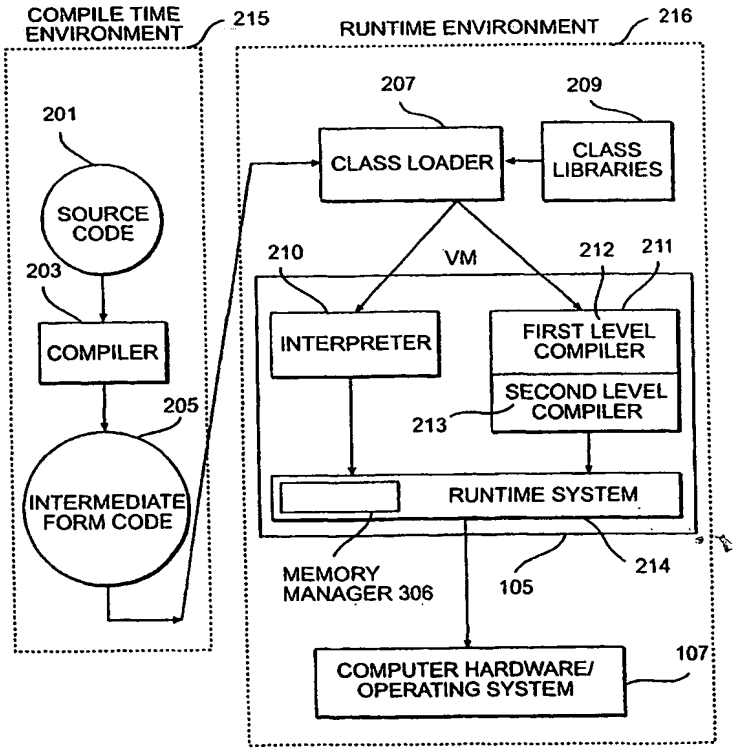
(81) Designated States: AE, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CU, CZ, DE, DK, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, UA, UG, UZ, VN, YU, ZA, ZW, ARIPO patent (GH, GM, KE, LS, MW, SD, SL, SZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

Published
Without international search report and to be republished upon receipt of that report.

(54) Title: FEEDBACK–BASED MEMORY ALLOCATION OPTIMIZATION IN A GARBAGE COLLECTION MEMORY MANAGEMENT SCHEME

(57) Abstract

A generational garbage collection system operates in conjunction with a run–time compiler to recompile methods that are determined, based on previous execution of methods in the system, to be long–lived so that the recompiled methods create objects directly into the older generation of the generational memory. The system uses a simple scheme to determine if an allocation site should be compiled as a long–lived allocation site, and may therefore be implemented in run–time in a multi–level compiler environment with relatively little modification to the existing compiler.



FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	**ES**	Spain	**LS**	Lesotho	**SI**	Slovenia
AM	Armenia	**FI**	Finland	**LT**	Lithuania	**SK**	Slovakia
AT	Austria	**FR**	France	**LU**	Luxembourg	**SN**	Senegal
AU	Australia	**GA**	Gabon	**LV**	Latvia	**SZ**	Swaziland
AZ	Azerbaijan	**GB**	United Kingdom	**MC**	Monaco	**TD**	Chad
BA	Bosnia and Herzegovina	**GE**	Georgia	**MD**	Republic of Moldova	**TG**	Togo
BB	Barbados	**GH**	Ghana	**MG**	Madagascar	**TJ**	Tajikistan
BE	Belgium	**GN**	Guinea	**MK**	The former Yugoslav Republic of Macedonia	**TM**	Turkmenistan
BF	Burkina Faso	**GR**	Greece			**TR**	Turkey
BG	Bulgaria	**HU**	Hungary	**ML**	Mali	**TT**	Trinidad and Tobago
BJ	Benin	**IE**	Ireland	**MN**	Mongolia	**UA**	Ukraine
BR	Brazil	**IL**	Israel	**MR**	Mauritania	**UG**	Uganda
BY	Belarus	**IS**	Iceland	**MW**	Malawi	**US**	United States of America
CA	Canada	**IT**	Italy	**MX**	Mexico	**UZ**	Uzbekistan
CF	Central African Republic	**JP**	Japan	**NE**	Niger	**VN**	Viet Nam
CG	Congo	**KE**	Kenya	**NL**	Netherlands	**YU**	Yugoslavia
CH	Switzerland	**KG**	Kyrgyzstan	**NO**	Norway	**ZW**	Zimbabwe
CI	Côte d'Ivoire	**KP**	Democratic People's Republic of Korea	**NZ**	New Zealand		
CM	Cameroon			**PL**	Poland		
CN	China	**KR**	Republic of Korea	**PT**	Portugal		
CU	Cuba	**KZ**	Kazakstan	**RO**	Romania		
CZ	Czech Republic	**LC**	Saint Lucia	**RU**	Russian Federation		
DE	Germany	**LI**	Liechtenstein	**SD**	Sudan		
DK	Denmark	**LK**	Sri Lanka	**SE**	Sweden		
EE	Estonia	**LR**	Liberia	**SG**	Singapore		

FEEDBACK-BASED MEMORY ALLOCATION OPTIMIZATION IN A GARBAGE COLLECTION MEMORY MANAGEMENT SCHEME

Background of the Invention

This disclosure relates generally to computers, and more particularly, to management of random access memory.

Computers generally use relatively fast random access memory (RAM) to store instructions and data during program execution. Despite the rapid growth in memory sizes of even the most modest computers, the supply of storage is not inexhaustible. It is a limited resource that must be allocated to programs. Some programming languages allow a programmer to manually allocate and reclaim memory for certain data. This data is said to be dynamically allocated by the programmer.

Manual reclamation of memory is often unsatisfactory, as it requires the programmer to keep track of and appropriately deallocate memory. Programming languages such as LISP and the Java™ programming language solve this problem by devolving responsibility for dynamic memory management to the program's run-time system. The programmer must still specifically allocate the storage, but she does not need to determine when that memory is no longer required because it is recycled automatically. This process —the automatic management of dynamically allocated storage— is called garbage collection.

One drawback associated with garbage collection systems is that the act of garbage collecting introduces computational overhead into the computer system. There have been numerous attempts to increase the efficiency of garbage collection algorithms. One such method of garbage collection is referred to as generational garbage collection.

G nerational garbage collection schemes divide the available memory area, called the heap, into two or more generations, segregating memory

objects by age. Objects are first allocated to the youngest generation, but are promoted (also called tenured) to older generations if they survive long enough. Based on the hypothesis that most objects die young, generational schemes concentrate their effort to reclaim storage on the youngest generation because it is there that most recyclable space is to be found. The younger generation is typically smaller than the older generation (e.g., 10 to 20 times smaller) and is collected more frequently.

In one conventional technique for improving the performance of a generational garbage collection system, described by David Cohn and Satinder Singh, "Predicting Lifetimes in Dynamically Allocated Memory," Maze et al., eds., Advances in Neural Information Processing Systems 9, MIT Press, the system observes running programs and attempts to determine object allocation call sites that tend to produce long-lived objects. The program is then manually recompiled using the observed knowledge so that the predicted long-lived objects, when created, are immediately tenured. If the life spans of the objects are correctly predicted, several advantages are gained: (1) the system does not waste time copying the objects between generations, and (2) the system does not waste time analyzing an object while it is in the younger generation. This technique has its drawbacks, however. For example, the learning algorithm of Cohn and Singh is computationally expensive, and hence cannot be used to adapt behavior while the program is running. Instead, the learning algorithm is trained by looking at previous runs of the program, then recompiled using what has been learned.

Thus, there is a need in the art to improve the efficiency of garbage collection schemes.

Summary of the Invention

According to a first aspect, the present invention includes a method for managing memory objects including the steps of: (1)providing longevity information about the allocation and deallocation of the objects in a run-time environment; (2) determining whether a first memory object is likely to be

long-lived based on the longevity information; and (3) compiling, in the same run-time environment, an allocation site for the first object that selects a generation for the first object based on the determination.

A second aspect of the present invention is a computer system comprising a number of elements, including: a compiler configured to compile instructions as the instructions are needed by the computer system for execution; a memory segmented into at least a younger generation and an older generation; a memory manager for automatically allocating and deallocating the memory in response to selected ones of the instructions that contain object allocation calls; and longevity database dynamically populated by the memory manager and including information relating to whether an object is likely to be a long-lived object. The compiler consults the longevity database before compiling the object allocation calls, and when the longevity database indicates that the object allocation calls are likely to create long-lived objects, the compiler compiles the indicated object allocation calls so that the object allocation calls create objects directly into the older generation of the memory.

An additional aspect of the present invention, related to the first aspect, is directed to a computer readable medium.

Brief Description of the Drawings

The accompanying drawings, which are incorporated in and constitute a part of this specification, illustrate several implementations consistent with this invention and, together with the description, help explain the principles of the invention. In the drawings,

Fig. 1 is a high-level block diagram illustrating interaction of hardware and software components in an exemplary computer system capable of executing
programs;

Fig. 2 is a block diagram illustrating a run-time environment for the execution of a program in a virtual machine;

Fig. 3 is a block diagram illustrating software entities within a virtual machine consistent with the present invention;

Figs. 4A and 4B are histograms of objects allocated at a particular allocation site;

Fig. 5 is a flow chart illustrating methods consistent with the present invention for tenuring objects and updating allocation site histograms;

Fig. 6 is a flow chart illustrating methods consistent with the present invention for compiling software methods; and

Fig. 7 is a flow chart illustrating methods consistent with the present invention for determining if an allocation site tends to produce long-lived objects.

Detailed Description

This disclosure describes a garbage collection system that operates in conjunction with a run-time compiler. The system observes a program's execution and determines, using a learning algorithm, whether memory objects tend to be long-lived or short-lived. The run-time compiler uses the observed information to dynamically adjust the compilation of object allocation sites so that when the sites create memory objects, the objects are placed directly into either the older or younger generation of the memory heap.

Referring to the accompanying drawings, a detailed description of the preferred embodiment will now be described. Wherever possible, the same reference numbers will be used throughout the drawings to refer to the same or like parts.

Methods and systems consistent with the present invention may be implemented in a virtual machine using a run-time compiler. An exemplary virtual machine is described in "The Java™ Virtual Machine Specification," Tim Lindholm and Frank Yellin, Addison Wesley (1997). Such a system will now briefly be described with reference to Figs. 1 and 2.

Fig. 1 is a high-level block diagram illustrating interaction of hardware and software components in an exemplary computer system capable of

executing programs written, for example, in the Java programming language. Computer 102 includes a computer hardware/operating system section 107 executing programs from memory section 103. Memory 103 can be a random access memory, and may additionally or alternatively include other storage media such as magnetic or optical disks.

Memory 103 stores one or more computer processes or programs 104a, 104b, and 104c. Computer processes 104a and 104b are programs comprised of intermediate form code, such as a bytecode, executing through virtual machine 105. The virtual machine is itself a process that when executed on computer 102, translates processes 104a and 104b into computer instructions native to computer hardware/operating system section 107. In this manner, virtual machine 105 acts as an interpreter for section 107. In contrast to processes 104a and 104b, process 104c uses instructions native to section 107, and thus does not use virtual machine 105 for execution.

Computer 102 may be connected via network 120 to additional computing resources. Signals traveling through network 120 can generally be referred to as "carrier waves," which transport information. Instruction for methods to implement the present invention can be transported via the carrier waves to the additional computing resources.

Fig. 2 is a block diagram illustrating compile time environment 215 and run-time environment 216 for the execution of a program by a virtual machine. Source code 201 is compiled by compiler 203 into intermediate form code 205, which is a more compact, computer representation of source code 201. Intermediate form code 205 is platform independent, which means that it is executable on any computer hardware system containing an appropriate virtual machine. Source code 201, intermediate form code 205, and compiler 203 may be located on computer hardware/operating system 107, or on a remote computer platform.

Before being executed by virtual machine 105, intermediate form code 205 is loaded by class loader 207, which may, as appropriate, integrate

selected class libraries 209 into the intermediate form code. Virtual machine 105 executes the loaded intermediate form code using either interpreter 210 or run-time compiler 211. Run-time compiler 211, may be, for example, a Just-in-time (JIT) compiler. Interpreter 210 converts the intermediate form code into appropriate native instruction code as it is required by runtime system 214, and ultimately, by section 107.

Runtime system 214 includes memory manager 306, which is responsible for allocation and deallocation of memory objects. Memory manager 306 will be described in more detail below with reference to Fig. 3.

As with interpreter 210, run-time compiler 211 is a program that converts intermediate form code into native instructions that can be sent directly to the computer hardware. However, while interpreter 210 converts the bytecode on an as-needed basis, run-time compiler 211 pre-compiles selected methods of the intermediate form code into code native to section 107. Using run-time compiler 211 will usually allow the code to run more quickly. This is particularly true if the compiled sections of the code are called multiple times.

Pre-compiling intermediate form code with run-time compiler 211 will not always increase the computer's speed. For example, because compiling intermediate form code is a relatively slow process, the system may initially experience a performance decrease. To alleviate this problem, run-time compiler 211 may be divided into a first level run-time compiler 212 and a second level run-time compiler 213. Generally, first level run-time compiler 212 performs a quick compilation of the intermediate form code. Intermediate form code that is frequently executed may then be further compiled by second level run-time compiler 213. Compilation in second level run-time compiler 213 requires more computational resources than the first level run-time compiler, but tends to produce more highly optimized code (i.e., code that runs faster).

Consistent with the present invention, run-time compiler 211, and optionally, interpreter 210, while executing processes, pay special attention to

object calls made by allocation sites within the processes, where an allocation site is defined as code that causes the creation of, when executed, a new object in the memory heap. More specifically, the run-time environment generates a longevity database based on calls to allocation sites. The longevity database is used to identify objects that tend to be long-lived.

As briefly stated above, an allocation site is defined as code that causes the creation of a new object in the memory heap. In the Java programming language, for example, an allocation site uses the *new* operator, as demonstrated in the code fragment: "Point APoint = new Point(11, 11)," which causes the run-time system to execute a method to create of an object of the "Point" type called "APoint."

Fig. 3 is a block diagram illustrating software entities within a Java virtual machine used in creation of longevity database 320. Object memory heap 302 is a generational memory heap divided into a younger generation, section 303, and an older generation, section 304. Each memory section contains memory objects such as objects 310, 312, 314, and 316. Objects placed in the younger generation are appended with an object identification code, such as object identifiers 311 and 313, which identify the allocation site in which the object was created. Further, objects in the younger generation contain an age field, such as age fields 321-323, which store how many scavenges (also referred to as garbage collections) the object has survived. Age field 321 is incremented by memory manager 306 when the object survives a scavenge. When creating new memory objects in section 303 of memory heap 302, memory manager 306 appends the identification code and the age field to the new object; preferably to the end of the object. Typically, heap 302 is implemented as a contiguous memory region, so lengthening the object to append the unique identification code or the age field involves incrementing the pointer an extra word. Alternatively, other methods of adding the identifier to the object can equivalently be used.

The identification code may be determined by using a program counter value at the allocation site (either a machine code program counter or

bytecode counter). Alternatively, the identification code can be a unique number pre-assigned to the allocation site.

In normal operation, objects, when initially created, are assigned to the younger generation 303. If an object remains in the younger generation long enough without being deleted during a memory scavenge operation, it is tenured by memory manager 306. Generation of memory objects in a generational memory, such as heap 302, and their subsequent deletion or tenure, is well known in the art. See, for example, Garbage Collection, by Jones Lins, pub. Wiley 1996. When objects are tenured to old section 304 of memory heap 302, their identification codes are removed and the fact that the object has been tenured is noted.

Longevity database 320 stores a histogram for each allocation site. Histograms for an exemplary allocation site are shown in Figs. 4A and 4B. Memory manager 306 uses the histograms to find objects that tend to be long-lived. Subsequent invocations of these types of objects are allocated directly to the older generation.

Histogram 402 plots the number of objects allocated at a particular site that have survived *i* scavenges, where $i = 0, 1, 2, \ldots, T$. T is the tenuring threshold: when an object has survived T scavenges, it is tenured. Histogram 402 is created by memory manager 306 based on the age field of the objects. Each time an object survives a scavenge, memory manager 306 increments its age field and updates histogram 402.

For example, suppose the allocation site corresponding to histogram 402 has allocated five objects since the last scavenge and ten prior to the last scavenge. As shown in histogram 402, of the ten objects allocated prior to the last scavenge, one has survived three scavenges, two have survived two scavenges, five have survived one scavenge, and the remaining two were reclaimed at the first scavenge after their allocation. Histogram 402 at i=0 has the value of seven, representing the two objects that were reclaimed at the first scavenge and the five newly allocated objects. Assume tenure in this system occurs when an object survives four scavenges (i.e., T=4). None of

the objects have been tenured yet.

Fig. 5 is a flow chart illustrating methods consistent with the present invention for tenuring objects and updating the allocation site histograms. Memory manager 306 generates a histogram for each allocation site, and, for each surviving object in younger generation 303, begins by storing the age field (step 501). The value of the object's histogram at *i* equal to the stored age is then decremented, (step 502), and the value at *i* equal to the age field plus one is incremented (step 503). Next, if the age field plus one is equal to the tenure value, memory manager 306 tenures the object (step 505). Otherwise, the age field of the object is incremented (step 506).

Assume an exemplary scavenge leaving: three of the previously created objects (by implication, two of the previously created objects have not survived); three objects surviving with a scavenge count of one; one object surviving with a scavenge count of two; and one object surviving with a scavenge count of three. Histogram 404, shown in Fig. 4B, is an updated version of histogram 402, after application of the method of Fig. 5 to the scavenged younger generation 303.

Consistent with the present invention, second level run-time compiler 213 checks the longevity database histograms before recompiling a method. If the histogram for an allocation site indicates that the allocation site has previously been responsible for many long-lived objects, the second level run-time compiler compiles the method containing the allocation site using an allocator that places the object directly into the older generation. Otherwise, second level run-time compiler 213 uses a standard allocator, which places the object into the younger generation.

Fig. 6 is a flow chart illustrating the method described in the previous paragraph in more detail. For each method to be compiled, second level run-time compiler 213 scans the method for allocation sites (step 602). Each allocation site, as previously discussed, is associated with a unique identification code that is used by compiler 213 to match the allocation site with its histogram in the longevity database (step 604). If the longevity

database indicates that the allocation site tends to produce long-lived objects, compiler 213 compiles the allocation site so that when executed, the allocation site creates a memory object stored directly in the older generation in the memory heap (steps 606, 608). Otherwise, second level run-time compiler 213 compiles the allocation site so that it creates a memory object in the younger generation (steps 606, 610).

Fig. 7 is a flow chart illustrating, in greater detail, the method of determining whether an allocation site tends to produce long-lived objects, as performed in step 606.

By monitoring prior collections of the older generation, memory manager 306 calculates the average cost (time) to reclaim an object from the older space (step 701). To generate this cost value, called Old_Cost, memory manager 306 notes how long collection of older generation 304 takes and divides this value by the number of bytes reclaimed in the collection.

In younger generation 303, a time penalty, T_s, may be associated with copying an object that survives a collection cycle or is tenured. An object that does not survive a single scavenge is reclaimed with essentially no time penalty. T_s, which relates to a single object, may be expanded to a total scavenging penalty for an allocation site, called Total_T_s, using the equation

$$Total\_T_s = T_s * (hist[1] + hist[2] + ... + hist[T]),$$

where hist[i] equals the value of the allocation site's histogram at age *i* (step 702).

The extra cost of reclaiming objects prematurely tenured from the older generation, Extra_Tenure, is calculated in step 703 as

$$Extra\_Tenure = Old\_Cost * N * (hist[0] + hist[1] + ... + hist[T-1]).$$

If Extra_Tenure is less than Total_T_s, then time is saved by allocating directly into the older generation (steps 704 and 705), while if Extra_Tenure is greater than Total_T_s, the system continues allocating into the younger generation (steps 704 and 706).

Although the above method was described in reference to allocation sites that produce fixed length objects, some allocation sites, such as sites allocating arrays, may produce objects of varying length. In this situation, instead of counting objects in the histogram and then multiplying by N to arrive at the total number of bytes, the system simply counts the number of allocated bytes. This may be achieved by correlating the size of each surviving object with its corresponding entry in the histogram.

The allocation site identification codes were previously described as being stripped from the object when it is tenured. However, by leaving the identification codes associated with the tenured objects, memory manager 306 can determine the "death rate" of objects in the older generation. Specifically, memory manager may use the identification codes in the older generation to keep a running total of the total number of objects tenured from an allocation site. The "death rate" may then be calculated as the total number of objects tenured minus the number of objects still present in the older generation, divided by the total number of objects tenured. Or, equivalently, the death rate equals

(number of tenured and reclaimed objects) / (number of tenured objects).

The death rate may then be used to refine the Extra_Tenure value as

Extra_Tenure' = (death rate) * (Extra_Tenure),

where Extra_Tenure' is the refined version of Extra_Tenure. Extra_Tenure' may optionally be used in steps 703 and 706 in place of Extra_Tenure to more accurately compare tenure and scavenging costs.

When leaving the identification codes associated with the tenured object, it is desirable, because it saves space, to strip the allocation site ID from older generation objects when the allocation site is recompiled. To do this, an additional bit may be added to each objects header. The bit is set when the object is tenured. When the method containing the allocation is recompiled (and hence the old version of the method is not going to be used anymore), its entry in the longevity database is marked as stale. When doing

an older generation garbage collection, any object from a stale allocation site has its allocation site identification code stripped, and the bit cleared. Accordingly, space is only used for the allocation site identification codes until the garbage collection following the recompilation of the method containing the allocation site. At the end of garbage collection of the older generation, space in the longevity database corresponding to the recompiled allocation site may also be reclaimed.

As described above, an improved garbage collection system observes allocation call sites in a generational memory scheme that are producing long-lived objects. When program methods are recompiled by the run-time system, allocation sites which tend to produce long-lived objects are coded to immediately tenure these objects to an older generation of the memory heap. The system uses a simple scheme to determine if an allocation site should be compiled as a long-lived allocation site, and may therefore be implemented during run-time in a multi-level (just-in-time) compiler environment with relatively little modification to the existing run-time compiler.

While there has been illustrated and described what are at present considered to be preferred embodiments and methods of the present invention, it will be understood by those skilled in the art that various changes and modifications may be made, and equivalents may be substituted for elements thereof without departing from the true scope of the invention. For example, while the foregoing systems and methods have been described with reference to a Java based run-time environment, other run-time environments could conceivably be used to implement the present invention. Additionally, although aspects of the present invention are described as being stored in memory, one skilled in the art will appreciate that these aspects can also be stored on or read from other types of computer-readable media, such as secondary storage devices, like hard disks, floppy disks, or CD-ROM; a carrier wave from the Internet; or other forms of RAM or ROM.

In addition, many modifications may be made to adapt a particular element, technique or implementation to the teachings of the present

invention without departing from the central scope of the invention. Therefore, it is intended that this invention not be limited to the particular embodiments and methods disclosed herein, but that the invention include all embodiments falling within the scope of the appended claims.

What Is Claimed:

1. A method for managing allocation of objects created by corresponding memory allocation sites in a system using generational memory management and having a younger generation and an older generation, the method comprising the steps of:

providing longevity information about the allocation and deallocation of the objects in a run-time environment;

determining whether a first memory object is likely to be long-lived based on the longevity information; and

compiling, in the same run-time environment, an allocation site for the first object that selects a generation for the first object based on the determination.

2. The method of claim 1, wherein the step of compiling further comprises the step of:

compiling an allocation site for the first object that causes the generation of the first object in the younger generation when the first memory object is determined to have a life-cycle that is not likely to be long-lived.

3. The method of claim 1, wherein the step of compiling further comprises the step of:

compiling an allocation site for the first object that causes the generation of the first object in the older generation when the first memory object is determined to be likely to be long-lived.

4. The method of claim 1, wherein the step of providing longevity information further includes the steps of:

storing identification codes for the allocation sites; and

generating, for each said identification code, a histogram relating to the number of surviving objects at each of *i* scavenges, where $i = 0, 1, \ldots, T$, where T is the tenure threshold.

5. The method of claim 1, further including the step of appending an identification code to ends of objects created in computer memory.

6. The method of claim 5, wherein the step of appending the

identification code further includes the step of setting the identification code equal to a value of a program counter.

7. The method of claim 5, wherein the step of appending the identification code further includes the step of setting the identification code equal to a unique number pre-assigned to the allocation site associated with the identification code.

8. The method of claim 4, further including the step of appending an age field to objects created in computer memory.

9. The method of claim 4, wherein the determining step further includes the step of determining that the first memory object is likely to be long-lived when a cost associated with reclaiming objects prematurely tenured from the older generation is less than a cost associated with scavenging from the younger generation.

10. A computer system comprising:

a compiler configured to compile instructions as the instructions are needed by the computer system for execution;

a memory segmented into at least a younger generation and an older generation;

a memory manager for automatically allocating and deallocating the memory in response to selected ones of the instructions that contain object allocation calls; and

a longevity database dynamically populated by the memory manager and including information relating to whether an object is likely to be a long-lived object;

wherein the compiler consults the longevity database before compiling the object allocation calls, and compiles the object allocation calls based on the result of consulting the longevity database.

11. The computer system of claim 10, wherein the compiler further comprises:

means for compiling the object allocation calls in the older generation of the memory when the longevity database indicates that the object allocation calls are likely to create long-lived objects.

12. The computer system of claim 10, wherein the compiler further comprises:

a first level compiler for compiling the computer instructions as the instructions are needed by the computer system; and

a second level compiler for recompiling the computer instructions that are executed frequently.

13. The computer system of claim 10, wherein the memory manager further comprises:

means for appending identification information to objects allocated into the younger generation of the memory; and

means for removing the identification information from the objects as the objects are tenured from the younger generation to the older generation

14. The computer system of claim 10, wherein the longevity database further includes an association of the identification information of the object to the number of times the object has been tenured.

15. A computer readable medium containing instructions for causing a computer that manages allocation of memory objects created by corresponding memory allocation sites under a generational memory management scheme that has a younger generation and an older generation to perform the steps of:

providing longevity information about the allocation and deallocation of the objects in a run-time environment;

determining whether a first memory object is likely to be long-lived based on the longevity information; and

compiling, in the same run-time environment, an allocation site for the first object that selects a generation for the first object based on the determination.

16. The computer readable medium of claim 15, wherein the step of compiling further includes instructions for causing the computer to perform the step of:

compiling an allocation site for the first object that causes the generation of the first object in the younger generation when the first memory object is determined to have a life-cycle that is not likely to be long-lived.

17. The computer readable medium of claim 15, wherein the step of compiling further includes instructions for causing the computer to perform the step of:

compiling an allocation site for the first object that causes the generation of the first object in the older generation when the first memory object is determined to be likely to be long-lived.

18. The computer readable medium of claim 15, wherein the step of providing longevity information further includes instructions for causing the computer to perform the steps of:

storing identification codes for the allocation sites; and

generating, for each said identification code, a histogram relating to the number of surviving objects at each of *i* scavenges, where $i = 0, 1, \ldots, T$, where T is the tenure threshold.

19. The computer readable medium of claim 15, further including instructions for causing the computer to perform the step of appending an identification code to ends of objects created in computer memory.

20. The computer readable medium of claim 19, wherein the step of appending the identification code further includes instructions for causing the computer to perform the step of setting the identification code equal to a value of a program counter.

21. The computer readable medium of claim 19, wherein the step of appending the identification code further includes instructions for causing the computer to perform the step of setting the identification code equal to a unique number pre-assigned to the allocation site associated with the identification code.

22. The computer readable medium of claim 16, further including instructions for causing the computer to perform the step of appending an age field to objects created in computer memory.

23. The computer readable medium of claim 15, wherein the determining step further includes instructions for causing the computer to perform the step of determining that the first memory object is likely to be long-lived when a cost associated with reclaiming objects prematurely tenured from the older generation is less than a cost associated with scavenging from the younger generation.

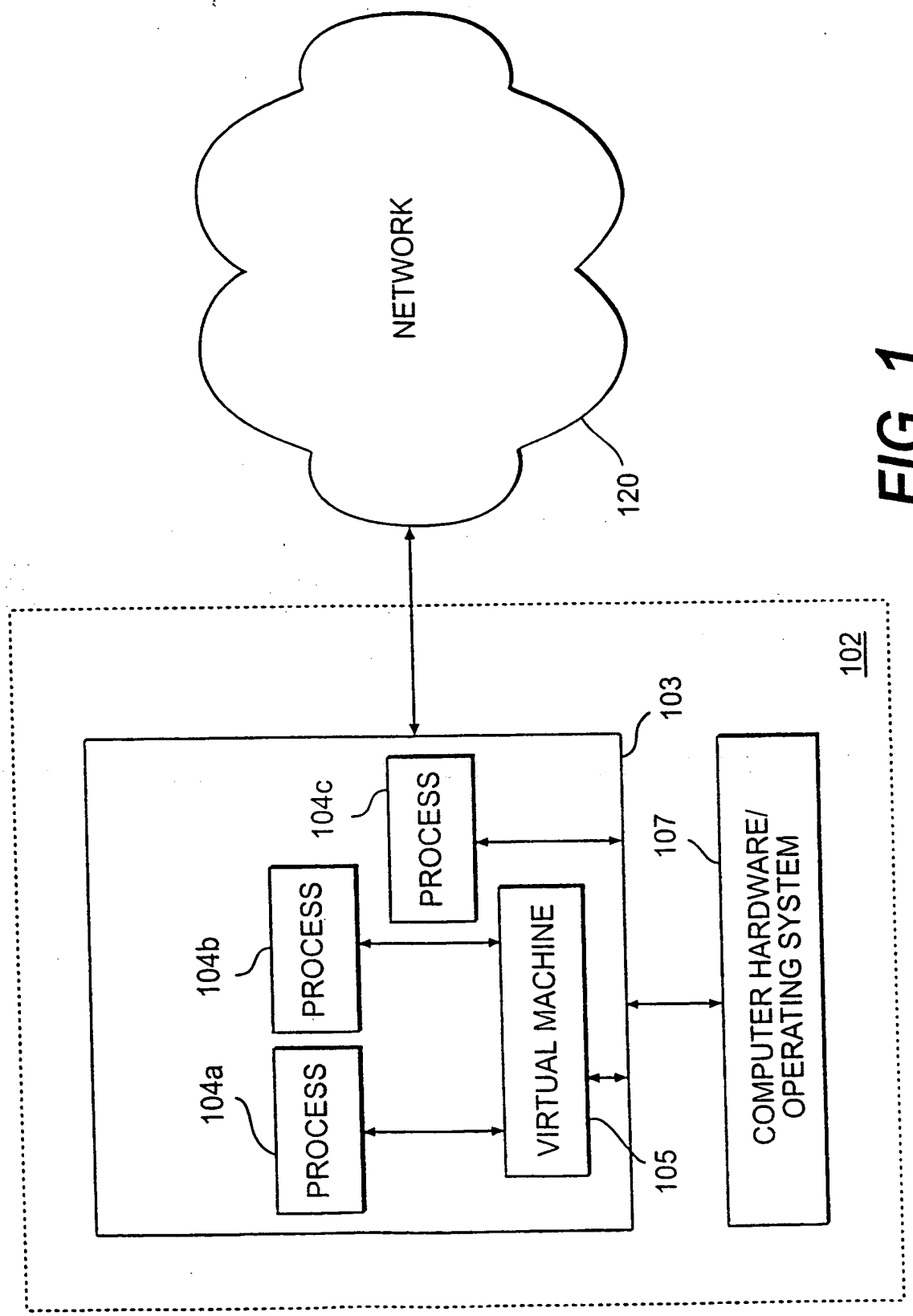


FIG. 1

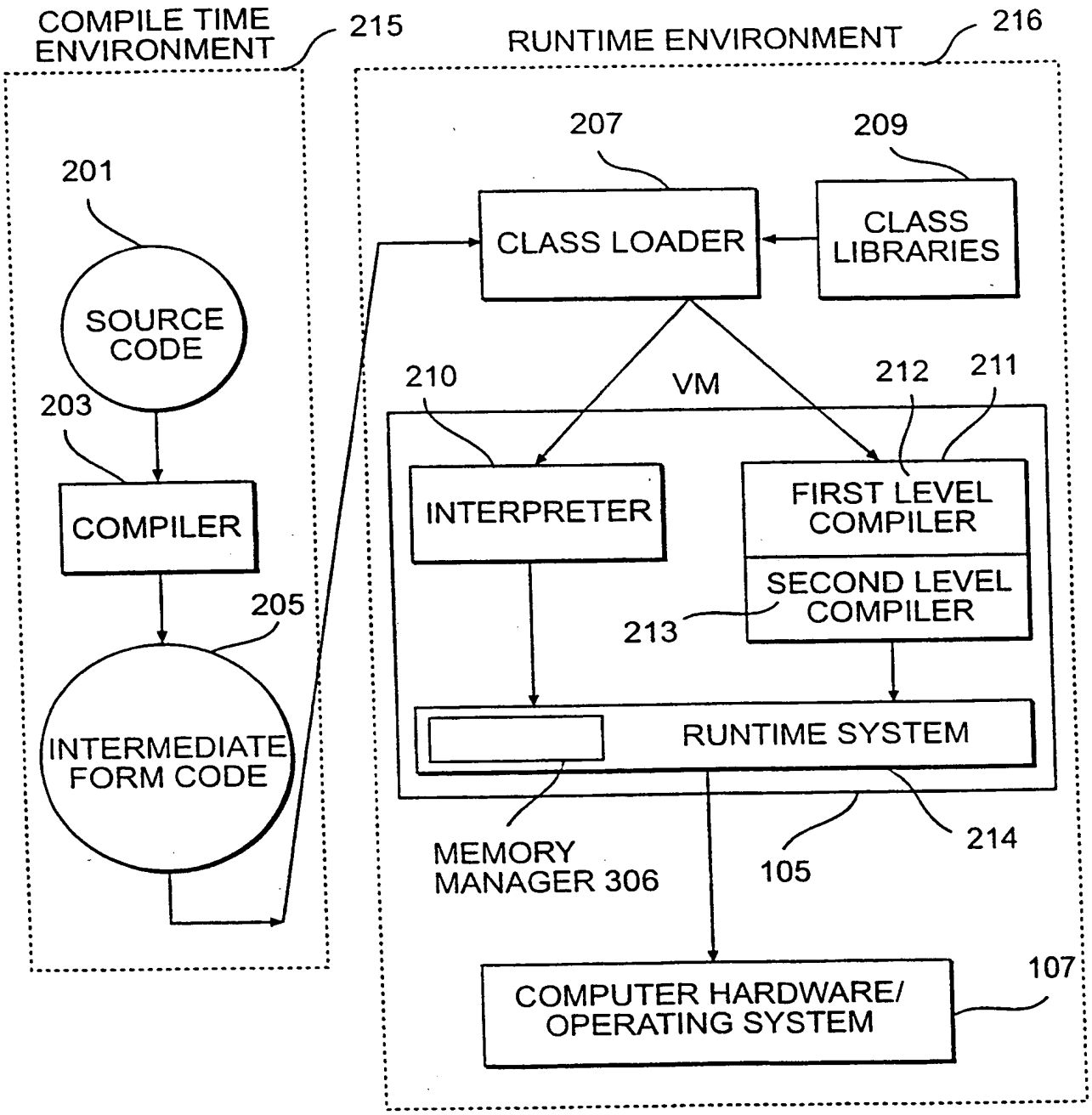

COMPILE TIME ENVIRONMENT
215
RUNTIME ENVIRONMENT
216
207
209
201
CLASS LOADER
CLASS LIBRARIES
SOURCE CODE
210
VM
212
211
203
INTERPRETER
FIRST LEVEL COMPILER
COMPILER
SECOND LEVEL COMPILER
205
213
RUNTIME SYSTEM
INTERMEDIATE FORM CODE
MEMORY MANAGER 306
105
214
107
COMPUTER HARDWARE/ OPERATING SYSTEM

FIG. 2

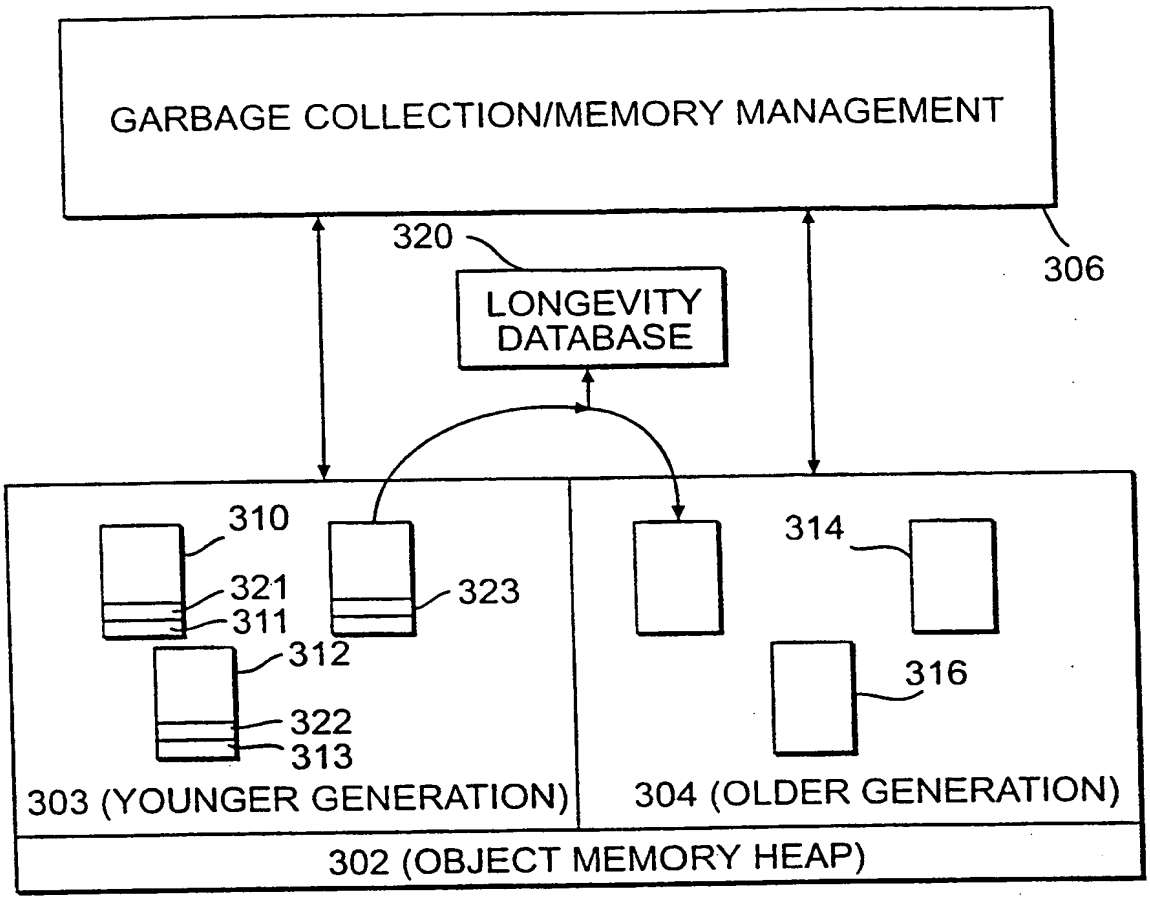

GARBAGE COLLECTION/MEMORY MANAGEMENT
320
LONGEVITY DATABASE
306
310
321
311
323
312
322
313
314
316
303 (YOUNGER GENERATION)
304 (OLDER GENERATION)
302 (OBJECT MEMORY HEAP)

FIG. 3

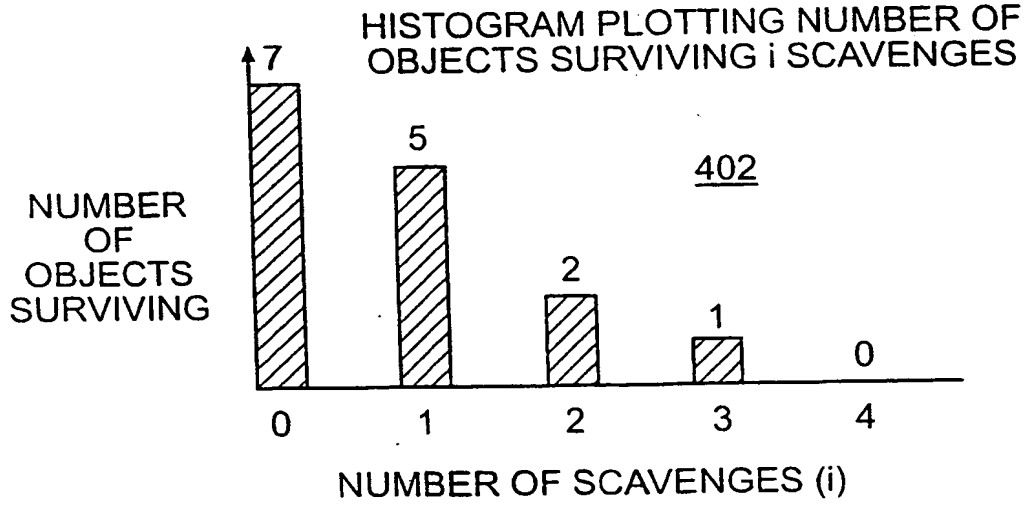

HISTOGRAM PLOTTING NUMBER OF OBJECTS SURVIVING i SCAVENGES
NUMBER OF OBJECTS SURVIVING
7
5
2
1
0
402
0
1
2
3
4
NUMBER OF SCAVENGES (i)

FIG. 4A

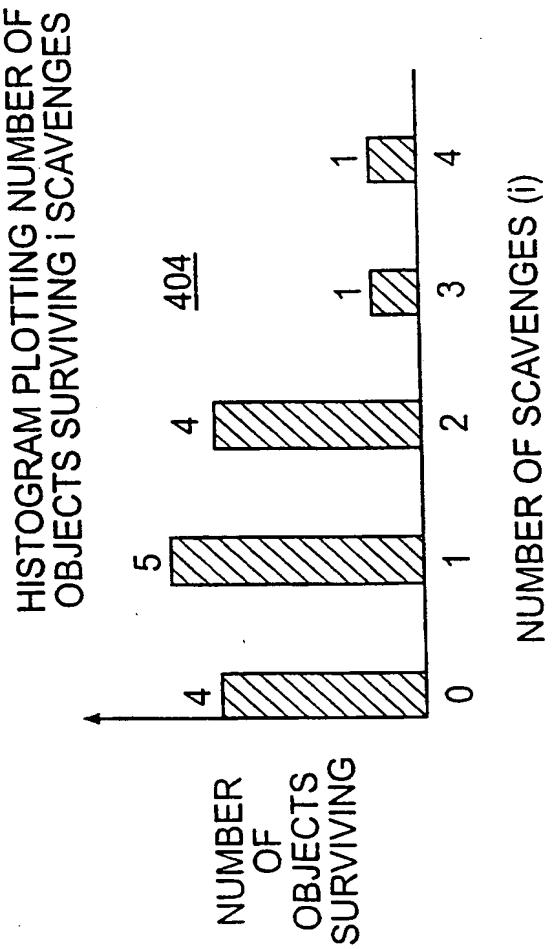

HISTOGRAM PLOTTING NUMBER OF OBJECTS SURVIVING i SCAVENGES
404
NUMBER OF OBJECTS SURVIVING
NUMBER OF SCAVENGES (i)
4
5
4
1
1
0
1
2
3
4

FIG. 4B

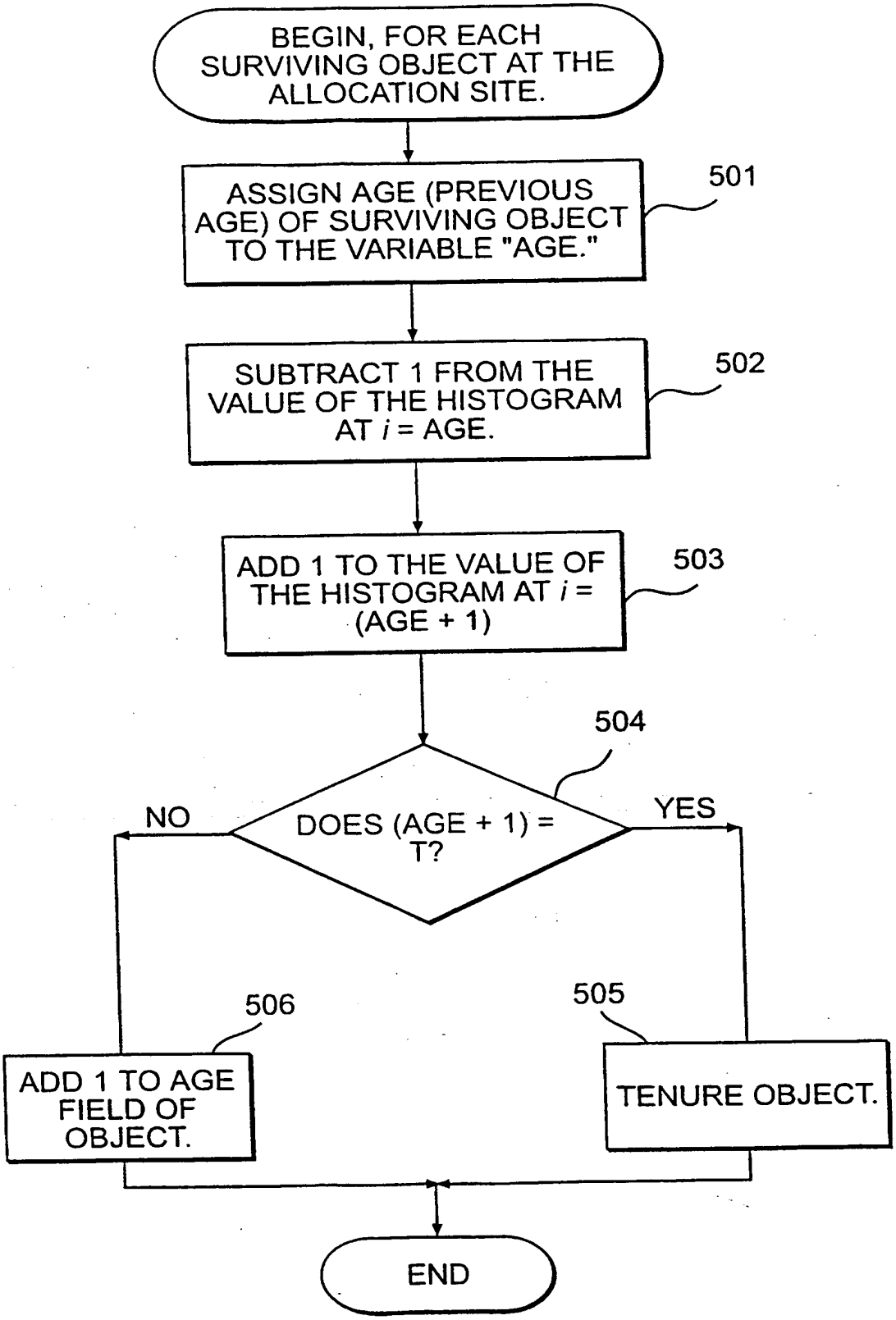

BEGIN, FOR EACH SURVIVING OBJECT AT THE ALLOCATION SITE.
ASSIGN AGE (PREVIOUS AGE) OF SURVIVING OBJECT TO THE VARIABLE "AGE."
501
SUBTRACT 1 FROM THE VALUE OF THE HISTOGRAM AT i = AGE.
502
ADD 1 TO THE VALUE OF THE HISTOGRAM AT i = (AGE + 1)
503
504
DOES (AGE + 1) = T?
NO
YES
506
ADD 1 TO AGE FIELD OF OBJECT.
505
TENURE OBJECT.
END

FIG. 5

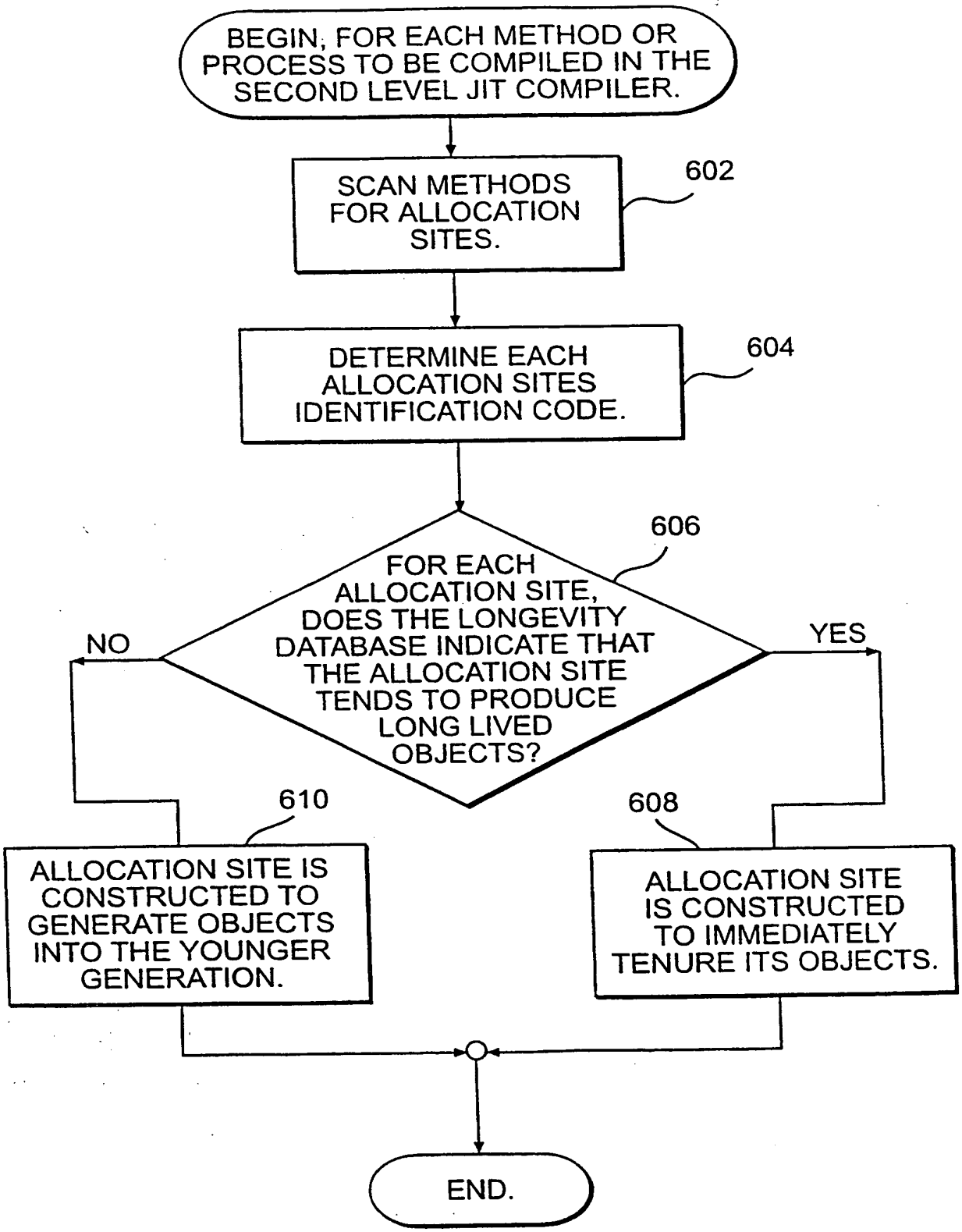

BEGIN, FOR EACH METHOD OR PROCESS TO BE COMPILED IN THE SECOND LEVEL JIT COMPILER.
SCAN METHODS FOR ALLOCATION SITES.
602
DETERMINE EACH ALLOCATION SITES IDENTIFICATION CODE.
604
FOR EACH ALLOCATION SITE, DOES THE LONGEVITY DATABASE INDICATE THAT THE ALLOCATION SITE TENDS TO PRODUCE LONG LIVED OBJECTS?
606
NO
YES
610
ALLOCATION SITE IS CONSTRUCTED TO GENERATE OBJECTS INTO THE YOUNGER GENERATION.
608
ALLOCATION SITE IS CONSTRUCTED TO IMMEDIATELY TENURE ITS OBJECTS.
END.

FIG. 6

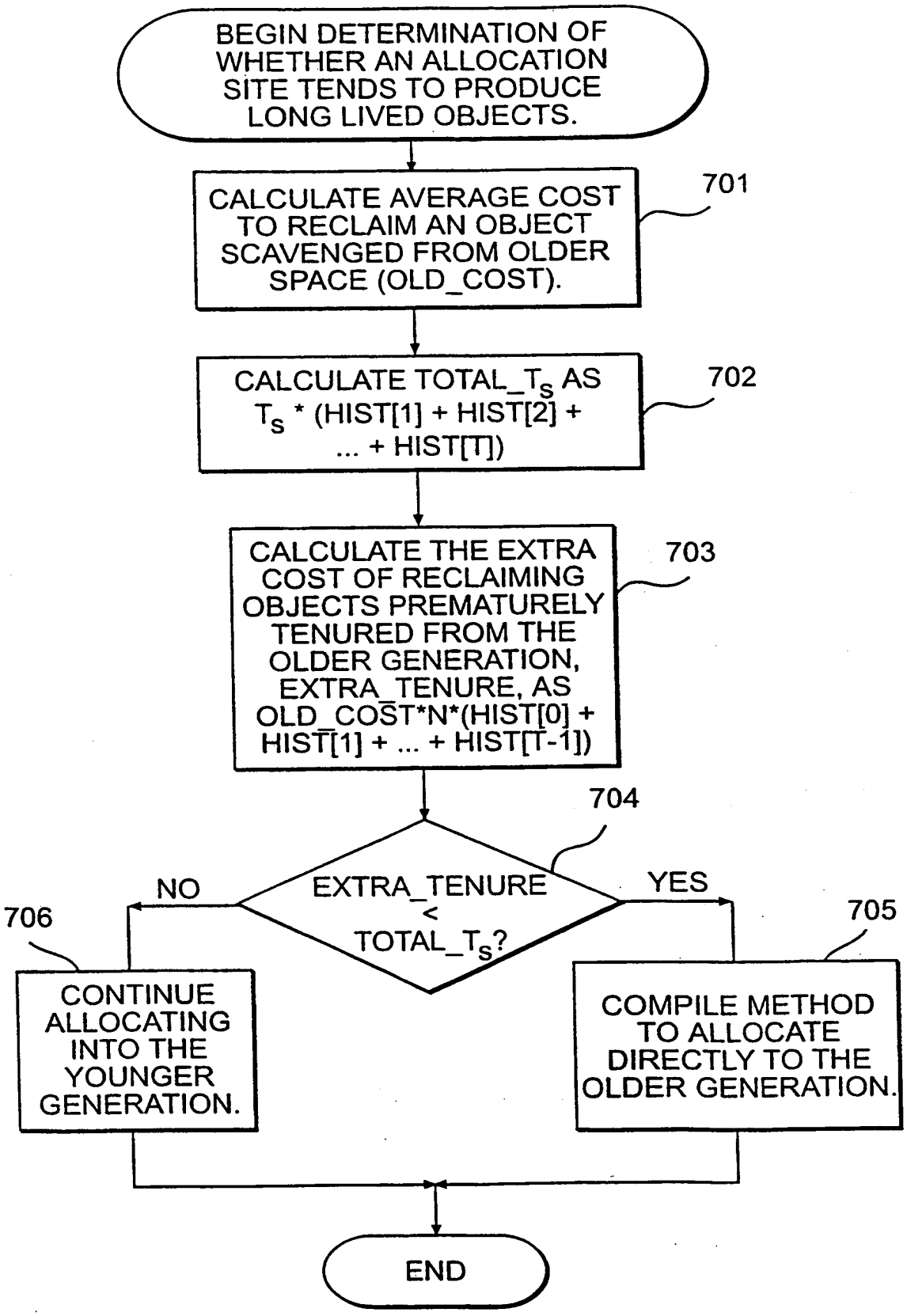


FIG. 7